JPA (Java Persistence API)

===========================

The Java Persistence API (JPA) is a specification of Java. It is used to persist data between Java object and relational database. JPA acts as a bridge between object-oriented domain models and relational database systems.

As JPA is just a specification, it doesn't perform any operation by itself. It requires an implementation. So, ORM tools like Hibernate, TopLink and iBatis implements JPA specifications for data persistence.

JPA Versions:

==============

The first version of Java Persistence API, JPA 1.0 was released in 2006 as a part of EJB 3.0 specification.

JPA 2.0 -

----

1. This version was released in the last of 2009.

2. It supports validation.

3. It shares the object of cache support.

JPA 2.1:

----

1. The JPA 2.1 was released in 2013

2. It allows fetching of objects.

3. It provides support for criteria update/delete.

4. It generates Schema

JPA 2.2:

----

1. The JPA 2.2 was released as a development of maintainenece in 2017

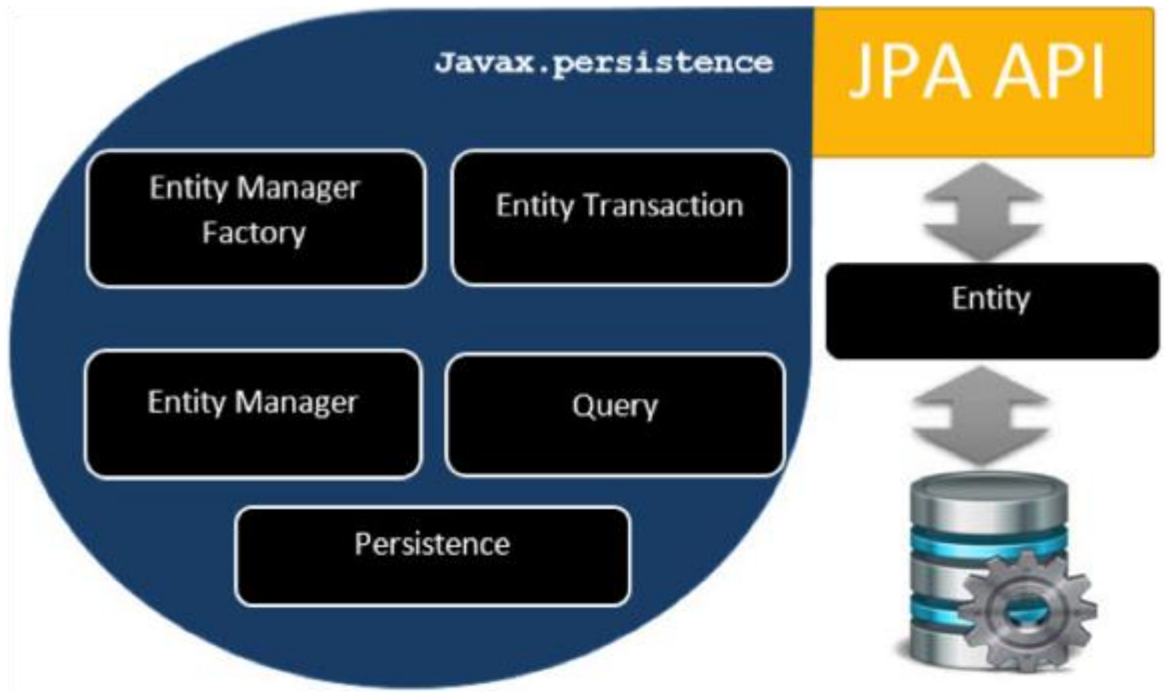
2. It supports Java 8 Date and Time.

3. It allows JPA annotation to be used in meta-annotations

4. It provides an ability to stream a query result.

JPA Architecture:

==============================



EntityManagerFactory: This is a factory class of EntityManager. It creates and manages multiple EntityManager instances.

EntityManager: It is an Interface, it manages the persistence operations on objects. It works like factory for Query instance.

Entity: Entities are the persistence objects, stores as records in the database.

EntityTransaction: It has one-to-one relationship with EntityManager. For each EntityManager, operations are maintained by EntityTransaction class.

Persistence: This class contain static methods to obtain EntityManagerFactory instance.

Query: This interface is implemented by each JPA vendor to obtain relational objects that meet the criteria.

JPA Object Relational Mapping:

--

Object Relational Mapping (ORM) is a functionality which is used to develop and maintain a relationship between an object and relational database by mapping an object state to database column. It is capable to handle various database operations easily such as inserting, updating, deleting etc.

ORM Frameworks:

Hibernate

TopLink

ORMLite

iBATIS

JPOX

JPA Entity:

================

Entities in JPA are nothing but POJOs representing data that can be persisted to the database. An entity represents a table stored in a database. Every instance of an entity represents a row in the table.

Entity Metadata:

Each entity is associated with some metadata that represents the information of it. Instead of database, this metadata is exist either inside or outside the class. This metadata can be in following forms: -

Annotation - In Java, annotations are the form of tags that represents metadata. This metadata persist inside the class.

XML - In this form, metadata persist outside the class in XML file.

We must have default constructor in Entity class.

Example:

```
@Entity

@Table(name = "employee")

public class Employee implements Serializable {


   private static final long serialVersionUID = -3009157732242241606L;


   @Id

   @GeneratedValue(strategy = GenerationType.AUTO)
```

```
    @Column(name="id")
    private long id;

    @Column(name = "firstname")
    private String firstName;

    @Column(name = "lastname")
    private String lastName;

    @Column(name = "age")
    private int age;

    protected Employee() {
    }
}
```

@Entity: Is used to define that the class is an Entity class.

@Table: This annotation is used for specifying the table name which is defined in the database.

@Id: Id annotation is used for specifying the Id attribute

@GeneratedValue: This is used when we want to set automatically generated value. GenerationType is the mechanism that is used for generating the value for the specific column.

@Column: This annotation is used for mapping the columns from the table with the attributes in the class.

JPA Entity Manager:

==========================

EntityManager is the primary JPA interface used by applications. Each EntityManager manages a set of persistent objects, and has APIs to insert new objects and delete existing ones. When used outside the container, there is a one-to-one relationship between an EntityManager and an EntityTransaction.

Entity manager is used to read, delete and write an entity.

Steps to persist an entity object:

--

The EntityManagerFactory interface present in java.persistence package is used to provide an entity manager.

```
EntityManagerFactory emf=Persistence.createEntityManagerFactory("Student_details");
```

createEntityManagerFactory () creates a persistence unit by providing the same unique name which we provide for persistence-unit in persistent.xml file

Obtaining an entity manager from factory.

```
EntityManager em=emf.createEntityManager();

em.getTransaction().begin();

em.persist(s1);

em.getTransaction().commit();

emf.close();

em.close();
```

Collection Mapping:

=======================

In JPA, we can persist the object of wrapper classes and String using collections.

List

Set

Map

JPA List Mapping:

```
@Entity

public class Employee {

   @Id

   @GeneratedValue(strategy=GenerationType.AUTO)

   private int e_id;
```

```
    @ElementCollection
    private List<Address> address=new ArrayList<Address>();
}
```

The annotation @ElementCollection represents the embedded object.

Now, create a class of embedded object Address.java. The annotation @Embeddable represents the embeddable object.

```
@Embeddable
public class Address {
 private int e_pincode;
 private String e_city;
}
```

```
//Persisting the Data
EntityManagerFactory emf=Persistence.createEntityManagerFactory("Collection_Type");
EntityManager em=emf.createEntityManager();
em.getTransaction().begin();
Address a1=new Address();
a1.setE_pincode(201301);
a1.setE_city("Noida");

Employee e1=new Employee();
e1.setE_id(1);
e1.getAddress().add(a1);

em.persist(e1);
em.getTransaction().commit();
em.close();
emf.close();
```

JPA Set Mapping:

Its same as List mapping

```
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int e_id;

    @ElementCollection
    private Set<Address> address=new HashSet<Address>();
}

@Embeddable
public class Address {
 private int e_pincode;
 private String e_city;
}
```

Get the Entity manage Object like List Mapping then commit the transcation.

JPA Map Mapping:

Its same as List mapping

```
@Entity
public class Employee {
   @Id
   @GeneratedValue(strategy=GenerationType.AUTO)
   private int e_id;

   @ElementCollection
   private Map<Integer,Address> map=new HashMap<Integer,Address>();
}

@Embeddable
public class Address {
 private int e_pincode;
 private String e_city;
}
```

Get the Entity manage Object like List Mapping then commit the transcation.

JPA Mapping Types:

=====================

One-To-One

The One-To-One mapping represents a single-valued association where an instance of one entity is associated with an instance of another entity.

The one-to-one association can be either unidirectional or bidirectional.

In unidirectional association, the source entity has a relationship field that refers to the target entity and the source entity's table contains the foreign key.

In a bidirectional association, each entity (i.e. source and target) has a relationship field that refers to each other and the target entity's table contains the foreign key. The source entity must use the mappedBy attribute to define the bidirectional one-to-one mapping.

unidirectional association:

--

Parent/Source:

@Entity

@Table(name = "instructor")

public class Instructor {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String firstName;

@OneToOne(cascade = CascadeType.ALL)

@JoinColumn(name = "instructor_detail_id")

private InstructorDetail instructorDetail;

}

Child/Destination

@Entity

@Table(name = "instructor_detail")

public class InstructorDetail {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

}

bidirectional association:

Parent/Source:

@Entity

@Table(name = "instructor")

public class Instructor {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String firstName;

@OneToOne(cascade = CascadeType.ALL, mappedBy = "instructor", fetch = FetchType.LAZY)

private InstructorDetail instructorDetail;

}

Child/Destination:

@Entity

@Table(name = "instructor_detail")

public class InstructorDetail {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@OneToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "instructor_id")

private Instructor instructor;

```
}
```

One-To-Many & Many-To-One

--

Unidirectional - In this type of association, only the source entity has a relationship field that refers to the target entity. We can navigate this type of association from one side.

Bidirectional - In this type of association, each entity (i.e. source and target) has a relationship field that refers to each other. We can navigate this type of association from both sides.

Unidirectional:

Parent/Source:

```
@Entity

@Table(name = "instructor")

public class Instructor {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int id;

@OneToMany(cascade = CascadeType.ALL)

private List < Course > courses = new ArrayList < Course > ();

}
```

Child/Destination:

```
@Entity

@Table(name = "course")

public class Course {

 @Id

 @GeneratedValue(strategy = GenerationType.IDENTITY)
 @Column(name = "id")
 private int id;
}
```

Bidirectional:

Parent/Source:

```
@Entity
@Table(name = "instructor")
public class Instructor {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private int id;
@OneToMany(mappedBy = "instructor", cascade = { CascadeType.All })
private List<Course> courses;
}
```

Child/Destination:

```
@Entity
@Table(name = "course")
public class Course {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 @Column(name = "id")
 private int id;
@ManyToOne(cascade = CascadeType.ALL)
@JoinColumn(name = "instructor_id")
```

private Instructor instructor;

}

Many-To-Many:

In many-to-many association, the source entity has a field that stores a collection of target entities. The @ManyToMany JPA annotation is used to link the source entity with the target entity

A student can like many courses, and many students can like the same course

As we know, in RDBMSs we can create relationships with foreign keys. Since both sides should be able to reference the other, we need to create a separate table to hold the foreign keys:

Such a table is called a join table. In a join table, the combination of the foreign keys will be its composite primary key.

Modeling a many-to-many relationship with POJOs is easy. We should include a Collection in both classes, which contains the elements of the others.

After that, we need to mark the class with @Entity and the primary key with @Id to make them proper JPA entities.

Also, we should configure the relationship type. So, we mark the collections with @ManyToMany annotations

Student Class

=================

@Entity

class Student {

@Id

Long id;

@ManyToMany

@JoinTable(

name = "course_like",

joinColumns = @JoinColumn(name = "student_id"),

inverseJoinColumns = @JoinColumn(name = "course_id"))

Set<Course> likedCourses;

}

Course Class:

===============

@Entity

class Course {

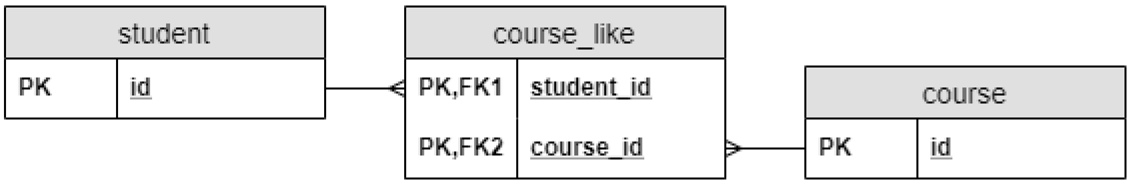
@Id

Long id;

@ManyToMany

Set<Student> likes;

}



https://www.baeldung.com/jpa-many-to-many

JPA Cascading Operations:

===========================

In JPA, if any operation is applied on an entity then it will perform on that particular entity only. These operations will not be applicable to the other entities that are related to it.

To establish a dependency between related entities, JPA provides **javax.persistence.CascadeType** enumerated types that define the cascade operations. These cascading operations can be defined with any type of mapping i.e. One-to-One, One-to-Many, Many-to-One, Many-to-Many.

PERSIST: In this cascade operation, if the parent entity is persisted then all its related entity will also be persisted.

MERGE: In this cascade operation, if the parent entity is merged then all its related entity will also be merged.

DETACH: In this cascade operation, if the parent entity is detached then all its related entity will also be detached.

REFRESH: In this cascade operation, if the parent entity is refreshed then all its related entity will also be refreshed.

REMOVE: In this cascade operation, if the parent entity is removed then all its related entity will also be removed.

ALL: In this case, all the above cascade operations can be applied to the entities related to parent entity.

Example:

@OneToOne(cascade=CascadeType.PERSIST)

private Subject sub;

JPA JPQL Introduction:

========================

The JPQL (Java Persistence Query Language) is an object-oriented query language which is used to perform database operations on persistent entities.

Instead of database table, JPQL uses entity object model to operate the SQL queries. Here, the role of JPA is to transform JPQL into SQL.

JPQL Features:

It is a platform-independent query language.

It is simple and robust.

It can be used with any type of database such as MySQL, Oracle.

JPQL queries can be declared statically into metadata or can also be dynamically built in code.

JPQL provides two methods that can be used to access database records:

@Query

@NamedQuery

@Query:

@Query can be applied on Methods. Its is used to prepare the custom queries.

A Query is similar in syntax to SQL, and it's generally used to perform CRUD operations

```
@Repository

public interface BookRepository extends CrudRepository<Book, Integer> {

  @Query("SELECT b FROM Book b")

List<Book> findAllBooks();

}
```

Native Queries:

Native query refers to actual sql queries (referring to actual database objects). These queries are the sql statements which can be directly executed in database

The @Query annotation allows for running native queries by setting the nativeQuery flag to true, as shown in the following example:

```
public interface UserRepository extends JpaRepository<User, Long> {

@Query(value = "SELECT * FROM USERS WHERE EMAIL_ADDRESS = ?1", nativeQuery = true)

User findByEmailAddress(String emailAddress);

}
```

The @Query annotation allows for running native queries by setting the nativeQuery flag to true, as shown in the following example:

@NamedQuery:

We define NamedQuery on the Entity class itself, providing a centralized, quick and easy way to read and find Entity‘s related queries.

The @NamedQuery annotation contains four elements - two of which are required and two are optional. The two required elements, name and query define the name of the query and the query string itself and are demonstrated above. The two optional elements, lockMode and hints, provide static replacement for the setLockMode and setHint methods.

Example:

```
@Table(name = "users")

@Entity

@NamedQuery(name = "UserEntity.findByUserId", query = "SELECT u FROM UserEntity u WHERE u.id=:userId")

public class UserEntity {

   @Id

   private Long id;

   private String name;

}
```

Update Queries With @Modifying:

==================================

We can use the @Query annotation to modify the state of the database by also adding the @Modifying annotation to the repository method.

Example:

```
@Modifying

@Query("update User u set u.status = :status where u.name = :name")

int updateUserSetStatusForName(@Param("status") Integer status,

  @Param("name") String name);
```

JPA Criteria API:

=========================

The Criteria API is one of the most common ways of constructing queries for entities and their persistent state. It is just an alternative method for defining JPA queries.

Criteria API defines a platform-independent criteria queries, written in Java programming language. It was introduced in JPA 2.0. The main purpose behind this is to provide a type-safe way to express a query

Steps to create Criteria Query:

Create an object of CriteriaBuilder interface by invoking getCriteriaBuilder() method on the instance of EntityManager interface.

EntityManager em = emf.createEntityManager();

CriteriaBuilder cb=em.getCriteriaBuilder();

Now, build an instance of CriteriaQuery interface to create a query object.

CriteriaQuery<StudentEntity> cq=cb.createQuery(StudentEntity.class);

Call from method on CriteriaQuery object to set the query root.

Root<StudentEntity> stud=cq.from(StudentEntity.class);

Methods of Criteria API Query Clauses:

--

SELECT: select()

FROM: from()

WHERE: where()

ORDER BY: orderBy()

GROUP BY: groupBy()

HAVING: having()

Note: - The CriteriaQuery interface is the sub-interface of AbstractQuery interface.

JPA Criteria SELECT Clause:

The SELECT clause is used to fetch the data from database. The data can be retrieved in the form of single expression or multiple expressions.

Generally, select() method is used for the SELECT clause to fetch all type of forms.

Example:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory( "Student_details" );

EntityManager em = emf.createEntityManager();

em.getTransaction().begin( );

CriteriaBuilder cb=em.getCriteriaBuilder();

CriteriaQuery<StudentEntity> cq=cb.createQuery(StudentEntity.class);

Root<StudentEntity> stud=cq.from(StudentEntity.class);

cq.select(stud.get("s_name"));
```

Selecting Multiple Expression:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory( "Student_details" );

EntityManager em = emf.createEntityManager();

em.getTransaction().begin( );

CriteriaBuilder cb=em.getCriteriaBuilder();

CriteriaQuery<StudentEntity> cq=cb.createQuery(StudentEntity.class);

Root<StudentEntity> stud=cq.from(StudentEntity.class);

cq.multiselect(stud.get("s_id"),stud.get("s_name"),stud.get("s_age") );
```

JPA Inheritance:

====================

Inheritence is a key feature of object-oriented programming language in which a child class can acquire the properties of its parent class. This feature enhances reusability of the code.

The relational database doesn't support the mechanism of inheritance. So, Java Persistence API (JPA) is used to map the key features of inheritance in relational database model.

JPA Inheritence Annotations:

==============================

@Inheritence - This annotation is applied on the root entity class to define the inheritance strategy. If no strategy type is defined with this annotation then it follows single table strategy.

@MappedSuperclass - This annotation is applied to the classes that are inherited by their subclasses. The mapped superclass doesn't contain any separate table.

@DiscriminatorColumn - The discriminator attribute differentiates one entity from another. Thus, this annotation is used to provide the name of discriminator column. It is required to specify this annotation on the root entity class only

@DiscriminatorValue - This annotation is used to specify the type of value that represents the particular entity. It is required to specify this annotation on the sub-entity classes.

Note - If we didn't pass the name of discriminator column and its value then JPA consider it by default. It considers DTYPE as discriminator column name and the name of the entity as discriminator value.

JPA Inheritance Strategies:

============================

- Single table strategy
- Joined strategy
- Table-per-class strategy

Single Table Strategy:

============================

The single table strategy is one of the most simplest and efficient way to define the implementation of inheritance. In this approach, instances of the multiple entity classes are stored as attributes in a single table only.

The following syntax represents the single table strategy: -

@Inheritance(strategy=InheritanceType.SINGLE_TABLE)

Example:

In this example, we will categorize employees into active employees and retired employees.

Thus, the subclass ActiveEmployees and RetiredEmployees inherits the e_id and e_name fields of parent class Employee.

Base Class:

```
@Entity

@Table(name="employee_details")

@Inheritance(strategy=InheritanceType.SINGLE_TABLE)

public class Employee implements Serializable {

@Id

private int e_id;

private String e_name;

}


//Sub Class

@Entity

public class ActiveEmployee extends Employee {


   private int e_salary;

   private int e_experience;

}


//Sub Class

@Entity

public class RetiredEmployee extends Employee {

   private int e_pension;

}
```

//Inserting the Data

```
EntityManagerFactory emf=Persistence.createEntityManagerFactory("Employee_details");

EntityManager em=emf.createEntityManager();

em.getTransaction().begin();

ActiveEmployee ae1=new ActiveEmployee(101,"Karun",10000,5);

RetiredEmployee re1=new RetiredEmployee(103,"Ramesh",5000);

em.persist(ae1);

em.persist(re1);

em.getTransaction().commit();

em.close();

emf.close();
```

Joined strategy:

=====================

In joined strategy, a separate table is generated for every entity class. The attribute of each table is joined with the primary key. It removes the possibility of duplicacy.

The following syntax represents the joined strategy: -

--

@Inheritance(strategy=InheritanceType.JOINED)

Example:

In this example, we will categorize employees into active employees and retired employees

Thus, the subclass ActiveEmployees and RetiredEmployees inherits the e_id and e_name fields of parent class Employee.

Base Class:

```
@Entity

@Table(name="employee_details")
```

```
@Inheritance(strategy=InheritanceType.JOINED)
public class Employee implements Serializable {
@Id
private int e_id;
private String e_name;
}
```

//Sub Class

```
@Entity
public class ActiveEmployee extends Employee {


   private int e_salary;
   private int e_experience;
}
```

//Sub Class

```
@Entity
public class RetiredEmployee extends Employee {
   private int e_pension;
}
```

//Inserting the Data

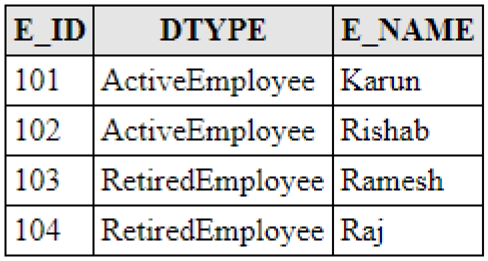
```
EntityManagerFactory emf=Persistence.createEntityManagerFactory("Employee_details");
EntityManager em=emf.createEntityManager();
em.getTransaction().begin();
ActiveEmployee ae1=new ActiveEmployee(101,"Karun",10000,5);
RetiredEmployee re1=new RetiredEmployee(103,"Ramesh",5000);
em.persist(ae1);
em.persist(re1);
```

em.getTransaction().commit();

em.close();

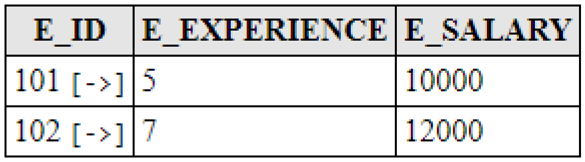
emf.close();

Select * from employee_details;



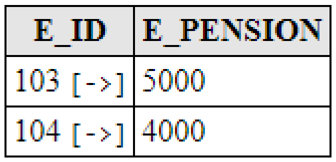
E_ID	DTYPE	E_NAME
101	ActiveEmployee	Karun
102	ActiveEmployee	Rishab
103	RetiredEmployee	Ramesh
104	RetiredEmployee	Raj

Select * from active_employee



E_ID	E_EXPERIENCE	E_SALARY
101 [->]	5	10000
102 [->]	7	12000

Select * from retired_employee



E_ID	E_PENSION
103 [->]	5000
104 [->]	4000

Table-per-class Strategy:

============================

In table-per-class strategy, for each sub entity class a separate table is generated. Unlike joined strategy, no separate table is generated for parent entity class in table-per-class strategy.

The following syntax represents the table-per-class strategy: -

----

@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)

In this example, we will categorize employees into active employees and retired employees.

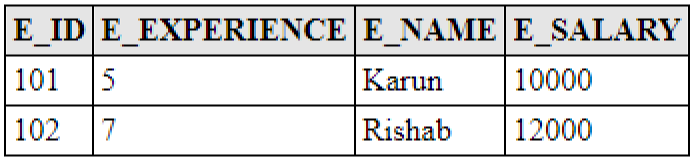
Thus, the subclass ActiveEmployees and RetiredEmployees inherits the e_id and e_name fields of parent class Employee.

Base Class:

```
------------
@Entity
@Table(name="employee_details")
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Employee implements Serializable {
@Id
private int e_id;
private String e_name;
}

//Sub Class
@Entity
public class ActiveEmployee extends Employee {


   private int e_salary;
   private int e_experience;
}

//Sub Class
@Entity
public class RetiredEmployee extends Employee {
   private int e_pension;
}

//Inserting the Data
```

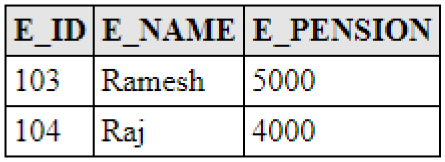
```
EntityManagerFactory emf=Persistence.createEntityManagerFactory("Employee_details");

EntityManager em=emf.createEntityManager();

em.getTransaction().begin();

ActiveEmployee ae1=new ActiveEmployee(101,"Karun",10000,5);

RetiredEmployee re1=new RetiredEmployee(103,"Ramesh",5000);

em.persist(ae1);

em.persist(re1);

em.getTransaction().commit();

em.close();

emf.close();
```

Select * from active_employee



E_ID	E_EXPERIENCE	E_NAME	E_SALARY
101	5	Karun	10000
102	7	Rishab	12000

Select * from retired_employee



E_ID	E_NAME	E_PENSION
103	Ramesh	5000
104	Raj	4000